Optimization Through Recomputation in the Polyhedral Model

Work in progress

Mike Jongen
Eindhoven University of Technology
Eindhoven, The Netherlands

Luc Waeijen
Eindhoven University of Technology
Eindhoven, The Netherlands

Roel Jordans*
Radboud RadioLab
Department of Astrophysics/IMAPP
Radboud University
Nijmegen, The Netherlands

Lech Jóźwiak
Eindhoven University of Technology
Eindhoven, The Netherlands

Henk Corporaal
Eindhoven University of Technology
Eindhoven, The Netherlands

Abstract

Many modern (mobile) systems involve memory intensive computations. External memory accesses are costly when it comes to the execution time and energy consumption of a program. To overcome this, we usually apply tiling to improve data locality and data reuse in internal memories. In the research reported in this paper we add the possibility to recompute data rather than storing temporary results, and demonstrate that this can have a positive effect on the overall application performance.

To achieve this we represented recomputation in the Polyhedral model by extending Polly. We experimentally verified the effectiveness of recomputation on a pair of Convolutional Neural Network layers, when applying loop tiling, loop fusion, and recompute.

Keywords Loop optimization, recompute, Polly

1 Introduction

Speed and energy consumption of program execution is important for all information processing systems, but they are of primary importance for mobile embedded systems with limited energy sources. Modern (mobile) systems increasingly exploit artificial vision, image processing, speech recognition, and similar data intensive processes that often involve artificial neural networks for their implementation. Recent examples are Google's Project Tango which enabled real-time 3d mapping from camera sensor data on a mobile phone and Apple's FaceID technology introduced in its latest iPhone models. Their algorithms involve a relatively large amount of memory accesses compared to the performed computations. Loop transformations are a powerful optimization technique enabling to substantially reduce the execution time and energy consumption of such memory intensive applications. Several popular transformations, such as loop tiling, fusion, and distribution, have already been modeled as transformations in the Polyhedral model. This allows for a more formal description of the transformations and usage of the Polyhedral transformation tools to ensure an optimized implementation without the risk of errors introduced by manual transformations.

So far the considered optimizations mainly focus on reordering of memory access patterns to increase the data locality, and thus cache performance or scratchpad utilization, of the target application. Polyhedral code generators such as CLooG [3] and isl [11] usually assume that statement schedules are single-valued, that is, any statement instance is executed only once in the transformed program. However, in some cases it has been shown that recomputing the result of a statement at a later time, as opposed to saving and reloading the result stored in memory, can be beneficial for the overall program performance [1]. In [9] an example of this recomputation has been demonstrated for data intensive applications such as Convolutional Neural Networks (CNN). The authors demonstrate that the energy consumption of running a CNN algorithm on a customized hardware accelerator architecture can be reduced by recomputing intermediate results. More recently [2] found that recomputation as optimization for CNNs is most effective for networks with small convolution kernels and the special class of recurrent-CNNs, both of which are becoming increasingly popular. A similar approach can be found in [6, 7], where the authors introduce overlapped tiling as a technique to enable improved parallelization of stencil computations over multi-core systems by introducing recomputation which is then distributed over the cores. In essence this technique applies the same transformation as we present here. Our focus however is the application of re-computation in single-core hardware accelerators where this technique can also help improving

*Also with Eindhoven University of Technology.

IMPACT 2018 (http://impact.gforge.inria.fr/impact2018)
Eighth International Workshop on Polyhedral Compilation Techniques,
In conjunction with HiPEAC 2018.
January 23, 2018, Manchester, United Kingdom

external memory access counts and local memory buffer requirements.

Currently usage of domain specific languages or manual code transformations are required to exploit recomputation, which hinders wide application and automated exploration of these design points. In order to enable automatic exploration of recomputation we modelled the recomputation transformation in the polyhedral model. However, the current loop transformation frameworks such as Polly [5] strongly build upon libraries like isl and have difficulty to work with statement schedules that include recomputation.

Our contributions in this paper are the following:

1. Formulate part of the polyhedral model representation of an example CNN application which includes recompute.
2. Extension of Polly to enable the automatic transformation of an imported schedule with recomputation into a single-valued variant, such that Polly can apply the transformation using the current infrastructure.
3. Demonstration of the effectiveness of recomputation on a single-core hardware accelerator to optimize the example application, by comparing against the optimized versions without recomputation and showing improved results.

The remainder of this paper is organized as follows. Section 2 presents a brief overview of the related work. Section 3 presents the example application and a short introduction to recomputation. Section 4 presents the modeling in the polyhedral model and the transformation of a schedule which includes recomputation into a single-valued schedule such that the existing transformation framework can be used. Section 5 presents the experimental evaluation of the work. Section 6 concludes the paper.

2 Related work

Convolutional neural networks have been gaining in strength over the past years and are now capable of handling many deep leaning tasks. However, the current generation of deep neural networks poses high requirements on the available memory bandwidth. Fortunately, the structure of these networks is often reasonably regular and is dominated by loop nests with static control [9]. Such *static control parts* (SCoPs) can be optimized reasonably well through modeling in the polyhedral model [10]. Considering recompute [2, 9] during the transformation can further improve the performance of the network, especially when it needs to run in an environment with reduced memory such as on an embedded platform.

Automated polyhedral optimization frameworks, such as Polly[5], R-Stream-TF [10], and PPCG [12] greatly reduce the effort of translating the original network description into an optimized form. Since these optimizations often require good bookkeeping of where data is stored they can actually guarantee the transformation is correct by automatically verifying the validity. Currently, none of these frameworks provides a method of including recomputation in the optimization space that we are aware of.

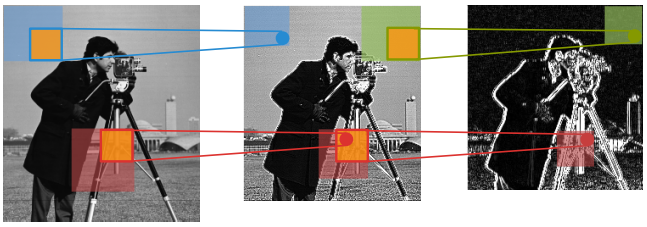

Figure 1. Two consecutive image filters. Tiling is illustrated at the top of the figure, indicated by the blue and green rectangles, and the orange 3*x*3 kernel. Fusion is illustrated in the bottom in red, where increasingly larger tiles are required in the preceding images to produce a tile in the output image.

For example, recent work in Polly has focused on improving tiling optimization heuristics based on local memory hierarchy information using the modeling from [8]. As a result Polly achieved a high performance increase on the generic matrix-matrix multiplication (GEMM) optimization getting much closer to the performance of hand optimized code.

In parallel to its own optimization heuristics, Polly also offers the ability to import a user-defined schedule for the application. This allows for the implementation of external optimization frameworks such as Pluto [4]. With the work presented here we aim at extending the polyhedral optimization frameworks with recomputation, initially by including recomputation as a possibility while importing externally optimized schedule through this interface. Finally, another key reason for selecting Polly over ClooG is its direct integration with the LLVM compiler framework which is used for our own low-power processor architectures (e.g. [13]). As such, choosing Polly allows for a stronger interaction between the optimization heuristics and final code generation parts of the compiler.

3 Optimizing through recompute

This section introduces the concept of recomputation, and explains how it can be useful. First an educational example is presented in Section 3.1. This example will be used throughout the work, and will be evaluated in more depth in the experiments presented in Section 5. Section 3.2 explains in more detail how the proposed recomputation compares to a method without recomputation.

3.1 Educational example

Instead of using a full-blown neural network with many dimensions, a reduced image processing pipeline is chosen

to explain and verify the concept of recomputation.[1] In this processing pipeline, two image filters are applied to an input image in sequence, as shown in Figure 1. For the example in this section, the image filters are assumed to be $3x3$. In the experiments in Section 5 different sizes will also be used. The observant reader might note that two consecutive convolutions can be mathematically combined into a single larger convolution. However, in convolutional networks this can not be done, because there usually are non-linear operations in between the convolution layers. Therefore we will have to assume that the convolutions can not be mathematically merged in our CNN example application and the algorithm itself will need to be optimized through loop fusion.

When this example application is to be mapped to an accelerator with limited local memory, the first strategy would be to tile the input. This is illustrated at the top of Figure 1. Tiling already improves the locality, but it can be improved even further by fusing the loop-nests of the two operations. Fusion matches the production of the intermediate image with it's consumption. Therefore the intermediate image does not need to exist completely at any point in time, allowing it to be kept in local buffers instead of larger external memories. This is illustrated using the red tiles in the lower half of Figure 1.

When fusion is applied, the tiles that are produced for the intermediate image can be kept in small local buffers, reducing accesses to higher memory levels. However, there will be some overlap between tiles in the intermediate layer depending on the kernel size. For this overlap there are two options. The pixels can either be stored in main memory for later use in a next tile, or they can be *recomputed* when they are needed for a later tile. Depending on the cost of accessing the external memory and the size of the internal buffer memory it can be beneficial for performance and/or energy consumption to recompute these pixels rather than to store them. This trade-off will be discussed in the remainder of this section.

There is one dimension that does not suffer from the tile overlap problem, which is the dimension the tiles "move" in. I.e., if two tiles that are adjacent in some dimension D are processed right one after the other, the overlap between the tiles will still be in the buffer when the second tile is processed. Hence the overlap does not need to be stored *or* recomputed. We refer to this dimension as the *inter tile dimension*. This effect is described in more detail by Peemen et al. [9]. The authors reason that because of this effect, the tile size in one selected dimension can be set to the full size of the input without increasing the required buffer space. In effect, any reuse in this dimension is obtained "for free", as there is no cost in extra storage space nor computations. In this work the tile size of the inter tile dimension is therefore always set to the full size of the input, and the storage versus recompute trade-off will be made for the remaining dimensions.

[1] This reduced image processing kernel is mainly used to increase the clarity of the presented method, the recompute optimization itself is not impacted by the simplification.

3.2 Recomputation tradeoffs

In this paper, three strategies of handling this overlap will be discussed:

1. **Global:** In the global strategy, the pixels overlapping between tiles are stored in external memory. This leads to a very small required local buffer size, but it also results in multiple expensive accesses to the main memory.
2. **Local:** The local strategy also stores the overlapping pixel, just like the global strategy. In the local strategy these pixels are stored in local buffers however, which results in larger internal buffers.
3. **Recompute:** The recompute strategy does not store the overlapping pixels, but rather recomputes them. This allows it to use the smaller internal buffers if the global strategy, combined with the reduced external accesses of the local strategy. This comes at a cost of an increased number of computations.

For each of these methods the code is restructured to create new intermediate buffers in the program, these buffers can then be mapped into either the global or local memory. This assignment process is currently part of our hardware modeling and is expected to be performed as part of the code generation. Careful matching of both the used local memory space and the actually available storage is performed as part of the hardware accelerator synthesis which allows for a fine-grain tuning of the provided amount of local storage.

The local method requires the least amount of external memory accesses, as every pixel only needs to be read once. The difference in external accesses for the other two strategies is more complex, and will be explained in more detail.

To better explain the trade-off between recomputation and store/load, a small example will be used which is illustrated using Figures 2 and 3. Figure 2 describes the situation when storing and loading the intermediate values and Figure 3 shows the situation when using recomputation. A kernel size of 3 will be used, and the output tile size is denoted as n. Both figures show the number of memory accesses required for the computation both from the local (cache or scratchpad) memory and from the external (global) memory.

When storing and loading the intermediate values, the first tile is larger than the other tiles. This is caused by the overlap between the tiles. This overlap is used by the first tile, and then no longer needs to be computed for the second tile. Both example figures show three tiles on the horizontal axis, for which the middle tile is most representative of the steady state of the loop-nest.

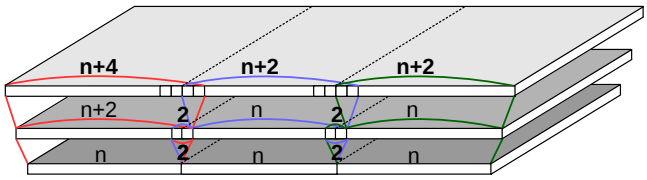

Figure 2. Traditional tiling of a fused loop-nest with storing of intermediate values. Tile size n, number of memory accesses shown, accesses in **bold** are from/to the global storage. Tiling requires storing of the intermediate value, memory access counts are otherwise equal

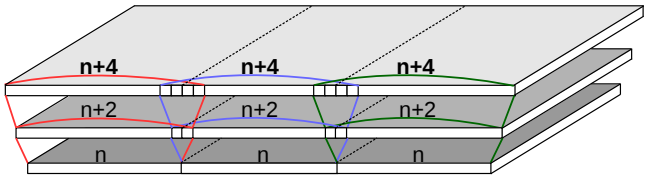

Figure 3. Tiling a fused loop-nest with recomputation of intermediate values. Tile size n, number of memory accesses shown, accesses in **bold** are from the global storage

In the case of traditional tiling (Figure 2), the first input tile has a size of $n+4$. This is used to calculate an intermediate tile of $n+2$ data values, which is subsequently used to calculate the first output tile. Next to the calculation of the output tile, the last 2 values of the intermediate tile need to temporarily be saved to memory for later usage. For subsequent tiles, the input tile moves to the right and only $n+2$ elements are loaded from the input tile as n new elements are computed for the intermediate layer, combining these n elements with the 2 previously stored values provides us with enough input data to compute the next n values of the output tiles. Combining this gives us $n+4$ loads for each computed output tile and 2 stores for saving the intermediate values (except for the last tile), in total $n+6$ accesses to the global storage for traditional tiling.

When using recomputation, every tile will have the same size, and perform the same computations. $n+4$ input elements are loaded per tile, which are each time used to compute an intermediate tile of $n+2$, which is then directly used to compute a output tile of size n. Since no elements are stored or loaded except for the input, only $n+4$ elements are loaded per tile in total.

From this example it can be concluded that fewer main memory accesses are required when using recomputation. Extrapolating for different kernel sizes shows us that the number of memory accesses saved will increase with increasing kernel sizes. This can be explained by the increased overlap between subsequent tiles (which need to compensate for the kernel dimension) and the related storage traffic in the traditional tiling scenario. This is counteracted by an increase in the number of instructions as the intermediate values now need to be recomputed. However, since main memory accesses can take thousands of cycles, recomputation can still result in a substantial performance increase, as will be shown in Section 5, even when complex computations are required to reproduce the intermediate results.

Finally, the presented recomputation can also be used for algorithms with higher dimensionality. However, the benefits of doing so can be expected to reduce significantly for each added dimension beyond 2d inputs as the number of recomputation steps will start increasing. Where for 1d inputs all inter tile overlap can be solved implicitly and in 2d input scenarios the amount of recompute scales approximately linearly with the kernel size. Extending this into 3d results in a quadratic relation between the kernel size and the amount of recomputation, thus reducing the expected benefits of recomputation.

4 Polyhedral modeling

Modeling recomputation in the polyhedral domain requires multiple executions of the statement computing the values for the intermediate layer within the tile overlap. This section introduces our method for modeling this recomputation within the polyhedral domain and how Polly was adapted to enable the exploration of recomputation as part of its code transformations.

4.1 The polyhedral model

The polyhedral model represents a program as a set of statements operating on data, the order of these statements is determined by their schedule. For example the map of statement:

$$Stmt[i0] \rightarrow [i0]$$

implies that the statement $Stmt$ with domain $i0$ is executed according to the increasing order of $[i0]$. This notation can be extended to multi-dimensional schedules to represent nested loops by introducing new variables to the domain and extending the schedule tuple.

Changing the schedule restructures the program execution similarly to applying a loop transformation. Legality of this transformation can be checked by verifying the data dependencies between the statement executions and optimized code may automatically be generated based on the transformed schedule. As a result, complex loop transformations can be reduced to schedule transformations in the polyhedral domain which helps ensuring the correctness of the transformed code.

4.2 Including recomputation

The main problem with recomputation however is that its simplest representation requires the assignment of multiple execution times to the computations of the intermediate results, such a schedule is known as a non-singular valued schedule. *A non-singular valued schedule* is a schedule for

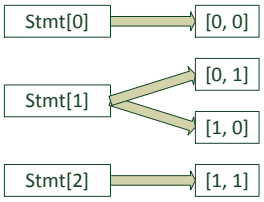

Figure 4. A graphical representation of the schedule of Stmt showing statements on the left and their schedule time on the right. $Stmt[1]$ is clearly showing a non-singular valued schedule entry as it is assigned two executions.

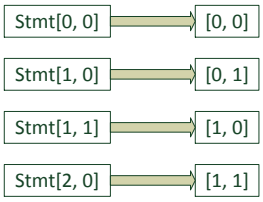

Figure 5. A graphical representation of the transformed schedule of the example showing the extended execution domain and resulting single valued schedule.

which one or more statement instances have multiple execution times assigned to them. The current polyhedral transformation tools currently explicitly avoid handling these kind of schedules, in both CLooG and ISL duplicates are automatically removed when an union of schedules is not disjoint.

There are two main solutions to this problem. Either the tools should be adjusted to be able to handle non-singular valued schedules, or the schedules themselves should be adjusted by introducing new statements, so that they are no longer single valued. Like Polly, many of the polyhedral transformation tools build upon the ISL library in order to handle the math behind the models. Within ISL there are a great number of mathematical functions that can be used but several of them assume a singular valued schedule. Simply inspecting them all and adjusting them to make them all work with non-singular valued schedules would be a huge undertaking, and, since the choice of ISL to provide no support for non-singular valued schedules is likely done on purpose, one would encounter a number of difficult (or maybe even impossible) mathematical problems. Therefore, it is better to stick to single-valued schedules. The remaining solution then is to implement a method to transform non-singular valued schedules to single valued systems when recomputation is found in the schedule.

To create single valued schedules, every iteration of every statement needs to be assigned an unique execution time. However, schedules using recomputation have statement iterations that have multiple execution times. For example, take the following (non-singular valued) schedule (which is visualized in Figure 4):

$$Stmt[i0] \rightarrow [t0, t1] : 0 \leq t0 < 2 \text{ and } 0 \leq t1 < 2 \text{ and } i0 = t0 + t1$$

Here the statement instance $Stmt[1]$ has two execution times ($[0, 1]$ and $[1, 0]$), making the schedule non-singular valued. To create a single valued version of this schedule (without changing the execution order of the schedule of course), $Stmt[1]$ should be executed at both $[0, 1]$ and $[1, 0]$. This can be done by making a copy of $Stmt$, and assigning

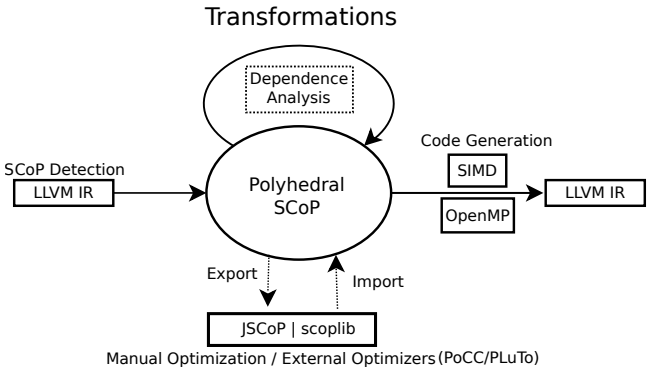

Figure 6. Structure of Polly within LLVM (from [5])

the original and the copy to $[0, 1]$ and $[1, 0]$ respectively. Resulting in the following schedules:

$$Stmt[i0] \rightarrow [0, t1] : t0 = 0 \text{ and } t1 = i0$$
$$Stmt'[i0] \rightarrow [1, t1] : t0 = 1 \text{ and } t1 = i0$$

However, when considering multiple recomputation instances of a single statement this can result in a large number of copied statements. This clutters the SCoP and makes analysis difficult. Therefore, the choice was made to add an iteration dimension to the execution domain:

$$Stmt[i0, t0] \rightarrow [t0, t1] : 0 \leq t0 < 2 \text{ and } 0 \leq t1 < 2 \text{ and } i0 = t0 + t1$$

Every execution time that lacks a unique iteration then is assigned a unique iteration using this dimension. This would work similarly to copying the statements, but keeps the "copies" organized in a single statement. For the example schedule, the result is given by Figure 5.

After the schedule modification method is chosen, the location of the transformation needs to be decided. Figure 6 shows the structure of Polly. Since Polly's JSCoP importer[2] itself already uses functions which requires singular-valued schedules, it is not possible to locate the transformation later than the importer itself. It would be possible to perform it earlier, and let the transformation be done during the adjustment of the JSCoP files. However, this would require

[2]Exported and imported schedules are stored in a JSON format by Polly.

more work in keeping the connection between the copied statements and the code they represent within the original program to perform the transformations correctly, which is not ideal. Moreover, it would still require adjustments to the importer, which currently cannot handle these kind of adjustments. Hence, the transformation was implemented in the JSCoP importer itself.

4.3 JSCoP Implementation

The first step in applying the transformations is to rewrite the SCoP of the input program so that it can be imported into Polly. Adding recomputation to a JSCoP schedule is done by assigning statement instances multiple execution times. For the examples in this paper, a convolution is tiled with overlapping tiles. This process works as follows, assume that an (untiled) 2D convolution has the following schedule:

$$Conv[i0, i1, i2, i3] \rightarrow [i0, i1, i2, i3]$$

The dimension $i0$ of this convolution is then tiled with a specified number of tiles no_tiles and $tilesize$, using two extra variables: $t0$ for the outer loop and $t1$ for the inner loop:

$$\begin{aligned} Conv[i0, i1, i2, i3] \rightarrow [t0, i1, t1, i2, i3] : \\ 0 <= t0 < no\_tiles \text{ and} \\ 0 <= t1 < tilesize \text{ and} \\ i0 = tilesize * t0 + t1 \end{aligned}$$

Recomputation is added to the schedule by increasing the domain of $t1$, without changing the definition of $i0$:

$$\begin{aligned} Conv[i0, i1, i2, i3] \rightarrow [t0, i1, t1, i2, i3] : \\ 0 <= t0 < no\_tiles \text{ and} \\ 0 <= t1 < tilesize + overlap \text{ and} \\ i0 = tilesize * t0 + t1 \end{aligned}$$

For $t0 = i$ and $tilesize <= t1 < tilesize + overlap$, and for $t0 = i + 1$ and $0 <= t1 < overlap$, $i0$ has the same value. This causes these elements to be computed multiple times. This method is used to add recomputation to a schedule by the examples in this paper, but all methods that add multiple execution times to an instance should be handled correctly by Polly.

4.4 Polly Implementation

After parsing through the JSON file, the JSON importer imports four parts: context, arrays, accesses and schedule. For the proposed extension, only the schedule part is relevant for detecting the recomputation. Polly uses the original (pre-transformation) SCoP when importing the new schedules. It checks if this new schedule satisfies the dependences of the old SCoP, and then adjusts the schedule of the old SCoP.

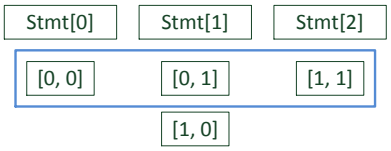

Figure 7. Lexicographical minimum of the example (highlighted in blue) demonstrating that $Stmt[1]$ has a non-singular valued schedule which requires adjustment.

However, transforming the schedule for recomputation requires that both the domain and accesses of the recomputed statements are also adjusted during cloning. When using recomputation, the new schedule does not simply satisfy the old dependencies. The cloned operations are placed on several locations in the schedule, and statements that depend on them require only one of these clones to be executed before them. This results in a complex situation when verifying the correctness of the transformation. Simply applying the old dependencies to the new schedule would result in unsatisfied dependencies, as not all of the cloned statements are executed before the statements that depend on them. Therefore, the duplication of operations requires the dependencies between the cloned operations and the successors to be updated. This has not yet been implemented, an therefore this legality check is disabled in the current implementation. The correctness of the program is currently checked by comparing the output to an unmodified execution of the program. Next, a check is executed to see if the new schedule is single valued. This way, single-valued schedules are not influenced by this adjustment. Before the actual implementation, a check is performed if the schedule is bound. An unbound schedule[3] would require infinite extra statements, and, since it does not represent an executable schedule, cannot be intentional. The check will cancel the importation if the schedule is unbound. The adjustment performs two tasks:

1. Adjustment of the old SCoP, so it can handle the new statement
2. Creation of the new schedule

The adjustments are performed statement by statement. The lexicographical minimum is used on the schedule. This gives us the lexicographically smallest execution time per iteration. Figure 7 shows the lexicographical minimum for the example schedule in blue. When a schedule is singular valued, its lexicographical minimum[4] is equal to the schedule itself. When it is not singular valued, it will give a singular valued map.

These properties are used to determine if the schedule of a statement is singular valued. Also see the pseudocode in

[3] An unbound schedule could be the result of a parametric kernel size which would require a compile time unknown amount of recomputation.
[4] Using the lexicographical maximum would give similar results.

Algorithm 1. If the lexicographical minimum of the schedule is equal to the original schedule, it is singular valued and it requires no further adjustments. If they are not equal, a new schedule needs to be created. At the start of creating this new schedule, a new dimension is added to the statement. Let us call this dimension $i1$ for now. The lexicographical minimum is then added to the new schedule for $i1 = 0$. By subtracting the lexicographical minimum from the original schedule, we are left with the part which is not yet in the new schedule. Of this part, the lexicographical minimum is taken again. This is then added to the new schedule for $i1 = 1$. A subtraction is performed, resulting again in the unscheduled part. This is continued until there is no unscheduled part left. The SCoP also needs to be adjusted. The domain of the adjusted statements is changed, by using the domain of the new schedule. The accesses are updated to work with the updated schedule. Finally, the new schedule is applied to the SCoP. This completes the transformation of the schedule with recomputation into a single valued schedule.

```
Data: OriginalSchedule
Result: NewSchedule
set Lexmin to the lexographical minimum of
  OriginalSchedule;
if OriginalSchedule is equal to Lexmin then
    set newSchedule to OriginalSchedule;
else
    add a dimension to newSchedule;
    set i to 0;
    RestofSchedule = OriginalSchedule - Lexmin;
    while RestofSchedule is not empty do
        add Lexmin to newSchedule with the new
          dimension set to i;
        i++;
        set Lexmin to the lexographical minimum of
          RestOfSchedule;
        RestofSchedule = RestofSchedule- Lexmin;
    end
end
```

Algorithm 1: How to make a schedule singular valued

5 Experimental results

This section described the experiments used to validate the use of recomputation, and it demonstrates how the proposed extensions of Polly are used to effectively explore the design space of different tile sizes and the use of recomputation versus storing the intermediate values either in global or local memory context.

For the first experiment the motivational example of Section 3.1 is taken. The input image is 480×320 pixels, and the tested tile sizes are powers of two ranging from 1 to 512. Because 480 is not a power of two, the tiles do not always nicely fit in the full width of the image, but since Polly is able with this, this does not pose a problem for our approach. Besides testing various tile sizes, the 3 main strategies of Section 3.2 are tested in this experiment.

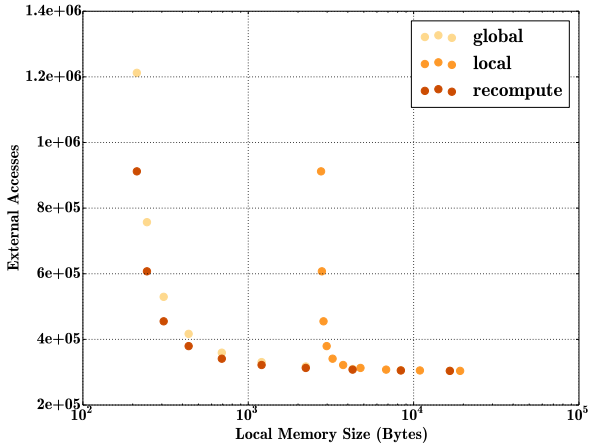

Figure 8. Memory accesses vs Required local buffer size for 2 consecutive convolutions with 3x3 kernel and different tile sizes

Using the debug printing of Polly a trace of the memory accesses is obtained for the various strategies and tile sizes. By also printing the memory locations of all the relevant arrays in the code, the loads and stores the all the arrays can be counted and assigned to the appropriate global/local memory ranges in the target architecture. Furthermore the size of the arrays is known, so by virtually mapping arrays to either local or global memory, it is possible to get the number of external accesses for each design point. These are plotted versus the buffer size in Figure 8 for a 3×3 kernel.

It is clear from Figure 8 that the three strategies follow the expectations. The global strategy requires only minimal internal buffer space, but has a high number of external accesses. The local strategy is the opposite of this, while the recompute strategy has the best of both worlds. In particular for the same internal buffer space, the recompute strategy can reduce the external accesses by almost 25% for small buffer sizes. As the internal buffer size grows, so does the size of the tiles that fit in it. The larger the tiles, the fewer tiles are needed to cover the entire image, thus for larger internal buffers, there is less overlap and the recompute strategy can gain less and less. This suggests recompute is particularly useful for situations where the available internal buffer space is relatively limited. This occurs for typical applications mapped to accelerators, or embedded platforms with small lower levels of cache. However, for applications with large input, such as big data processing, again many tiles will be needed even if the internal buffer size is rather large. Also in these cases recompute can reduce the accesses to

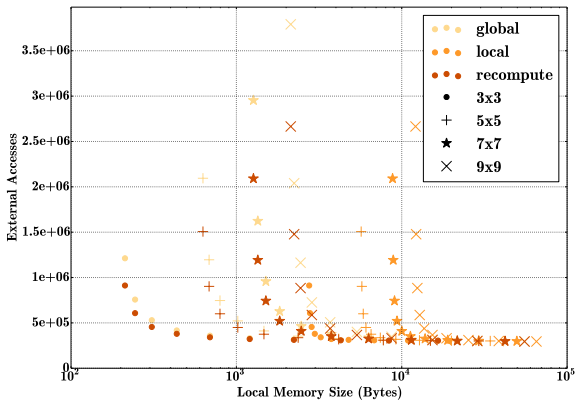

Figure 9. Memory accesses vs Required local buffer size for 2 consecutive convolutions for different kernel and tile sizes

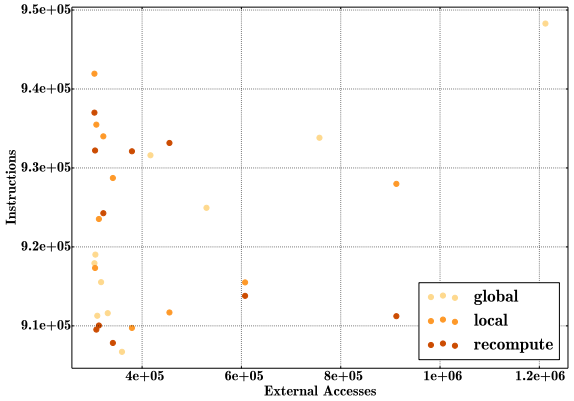

Figure 10. The number of external accesses vs the issued instructions for a 3×3 convolution window as measured by perf

external memory, improving performance depending on the characteristics of the application.

The amount of overlap between two tiles is determined by the size of the convolutional kernel. In the educational example, only a 3×3 filter is used, but often larger kernels are used in real applications. Since the overlap increases for larger kernel sizes, it is expected that the gain of recomputation over the local and global strategies will increase even further. As a test, experiments were performed with 4 different kernel sizes, i.e., 3×3, 5×5, 7×7 and 9×9. The results of these experiments are summarized in Figure 9.

From Figure 9 it is clear the shape of the curves of different kernel sizes matches those of the curves of the 3×3 kernel in Figure 8. However, as expected the gains increase even further for larger kernel sizes. Already for the 7×7 kernel, the number of external accesses can be reduced by 29% in the most extreme case.

Of course the gains in required internal memory and external accesses come at the cost of increased computational workload. For modern platforms computation is orders of magnitude faster and cheaper in energy than accessing higher levels of memory, so it is a trade-off that can be very interesting. To measure the impact of the recompute strategy, the tool *perf* was used to measure the number of issued instructions of all strategies as executed on a PC platform. The results are shown in Figure 10. Interestingly, the recompute strategy does not consistently require more computations than the other strategies in these measurements. Most likely this is because the control flow of the recompute strategy is less complex. When memory accesses are avoided, so are the computations required to accesses them. This very well might balance out the added cost of the increased computational work. It also would explain why the local strategy has reasonably high instruction counts, since the addressing into local buffers can be quite computational intensive. However, more investigation is needed to confirm this is indeed why the recompute strategy seems to have such a low cost in overall computation compared to the other strategies.

6 Conclusion & future work

In this work we have modelled recomputation in the polyhedral model. This modelling has been implemented in Polly, to both verify the model and ultimately make this optimization easily available to anyone using the LLVM compiler framework. The functionality of our implementation is verified using an example vision pipeline, which by extension can be turned into a complete convolutional neural network to which our model also would apply. The benefits of recomputation are shown in the experimental results section, for various convolution kernel sizes, and different internal buffer sizes. The ability to automatically generate code for all these points enables developers to more quickly evaluate the design space without the need for manual code transformations.

After this first proof-of-concept, still plenty of work remains for future work. One of the key steps will be to reenable the legality checks Polly performs when importing a modified SCoP, to give people more confidence in the transformations, and enable acceptance of our modifications in the main Polly release. Furthermore we plan to also model the effects of the recomputation transformations before code generation. Such a model can be used to automatically, and very quickly, explore the design space, ultimately allowing polly to automatically select an optimal transformation given some cost function for a specific back-end target. Also more applications need to be tested, including Deep Neural Networks which can potentially tremendously benefit from the recompute versus store trade-off.

Acknowledgement

The authors would like to thank the people at the Polly mailinglist for their helpful remarks during the development of our Polly extension. We also thank the reviewers for their helpful and detailed remarks and pointing us to the work on overlapped tiling. This research has received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under grant agreement no 676240 (NeMeCo).

References

[1] Ismail Akturk and Ulya R. Karpuzcu. 2017. AMNESIAC: Amnesic Automatic Computer Trading Computation for Communication for Energy Efficiency. In *ASPLOS 2017 - 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 811–824.

[2] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[3] Cédric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*. Juan-les-Pins, France, 7–16.

[4] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. 2008. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *International Conference on Compiler Construction (ETAPS CC)*.

[5] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly – performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 04 (2012), 1250010.

[6] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-Performance Code Generation for Stencil Computations on GPU Architectures. In *ICS '12 Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 311–320.

[7] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2007. Effective Automatic Parallelization of Stencil Computations. In *PLDI '07 Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 235–244.

[8] Tze Meng Low, Francisco D Igual, Tyler M Smith, and Enrique S Quintana-Orti. 2016. Analytical modeling is enough for high-performance BLIS. *ACM Transactions on Mathematical Software (TOMS)* 43, 2 (2016), 12.

[9] Maurice Peemen, Bart Mesman, and Henk Corporaal. 2015. Inter-tile reuse optimization applied to bandwidth constrained embedded accelerators. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*. IEEE, 169–174.

[10] Benoıt Pradelle, Benoıt Meister, Muthu Baskaran, Jonathan Springer, and Richard Lethin. 2017. Polyhedral Optimization of TensorFlow Computation Graphs. In *6th Workshop on Extreme-scale Programming Tools (ESPT-2017) at The International Conference for High Performance Computing, Networking, Storage and Analysis (SC17)*.

[11] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model.. In *ICMS*, Vol. 6327. Springer, 299–302.

[12] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4 (Jan. 2013), 54:1–54:23. https://doi.org/10.1145/2400682.2400713

[13] Luc Waeijen, Dongrui She, Henk Corporaal, and Yifan He. 2013. SIMD made explicit. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*. IEEE, 330–337.